CS 112

Introduction to Data Structures

WEEK 09

Stacks, Queues, and Hash Tables

Eric Araújo

Calvin University · Fall 2026

This Week

1. Abstract Data Types
2. The Stack ADT: LIFO
3. The Queue ADT: FIFO
4. Why the naive array queue fails
5. Circular arrays **KEY**
6. Hash tables: computing the index
7. Collisions and load factor

Abstract Data Type

An ADT is a promise about **behaviour**, with no commitment about **storage**.

The ADT says

- what operations exist
- what they mean
- what they cost

The ADT doesn't say

- array or linked list
- where the front is
- how it grows

> Interface versus implementation, the same idea as week 3's `public`/`private`, now applied to a whole data structure.

The Stack: Last In, First Out

```
push(item)   // put on top
pop()        // take off top
peekTop()    // look, don't take
isEmpty()
getSize()
```

top →	white
green	
blue	

A stack of plates. Push green, push white, pop; you get **white** back, because it was last on.

Stacks Are Everywhere

- **Ctrl-Z**: the last change is the first undone
- **Browser back button**: the page you just left is on top
- **The run-time stack**: functions return in reverse call order
- **Matching brackets**: push an opener, pop when you meet its closer

> Whenever "most recent first" is the right answer, a stack is the right structure. You'll meet the run-time stack again next week, in recursion.

TALK TO YOUR NEIGHBOR · TTYN

What does this print?

```
s.push(7);
s.push(8);
s.push(9);
while (!s.isEmpty()) {
    cout << s.pop() << " ";
}
```

A. 7 8 9

B. 9 8 7

C. 9 7 8

D. 7 9 8

Implementing a Stack

Array: top at the END

```
void push(const Item &it) {
  if (isFull()) throw ...;
  myArray[mySize] = it;
  ++mySize;
}
```

O(1). Top at index 0 would shift everything, O(n).

Linked list: top at the FRONT

```
void push(const Item &it) {
  myList.prepend(it);
}
```

O(1). Top at the back would need a walk to find it, O(n).

Same ADT, two implementations, one design rule: **put the action where the structure is fast.**

TALK TO YOUR NEIGHBOR · TTYN

Implementing a stack with a linked list, where should the top be?

A. At the front, prepend and remove-first are both O(1)

B. At the back; that's where append is

C. Either; both are O(1)

D. In the middle, to balance the cost

The Queue: First In, First Out

```
add(item)       // join the back
remove()        // serve the front
peekFront()
isEmpty()
getSize()
```

`front          11 → 22 → 33          back`

Serve at the front, join at the back, a supermarket queue.

> Also called First Come, First Served. Print jobs, request handling, breadth-first search, all queues.

The Obvious Array Queue Is Wrong

Front at index 0, back at `mySize - 1`. Adding is easy. Removing?

```
before          [ 11 | 22 | 33 | _ ]
remove()        [ 22 | 33 | _ | _ ]   everything shifted down
```

Every `remove()` shifts every remaining item: **O(n)**. For a structure whose whole job is serving the front, that's a bad trade.

Don't Move the Data, Move the Ends

index	[0]	[1]	[2]	[3]	[4]
data	_	22	33	44	_
		head = 1		tail = 3	

Keep two indices. `remove()` just moves `head` forward, no shifting, **O(1)**.

...And Wrap Around the End

index	[0]	[1]	[2]	[3]	[4]
data	66	_	33	44	55
	head = 2		tail = 0		(wrapped)

```
tail = (tail + 1) % myCapacity;     // advance with wraparound
head = (head + 1) % myCapacity;
```

> The modulo turns the array into a **circle**. Both ends move; nothing is ever copied.

TALK TO YOUR NEIGHBOR · TTYN

With `head == tail`, is the circular queue empty or full?

A. Empty

B. Full

C. Ambiguous; you can't tell without more information

D. Impossible; they can never be equal

Circular Array: The Trade

Wins

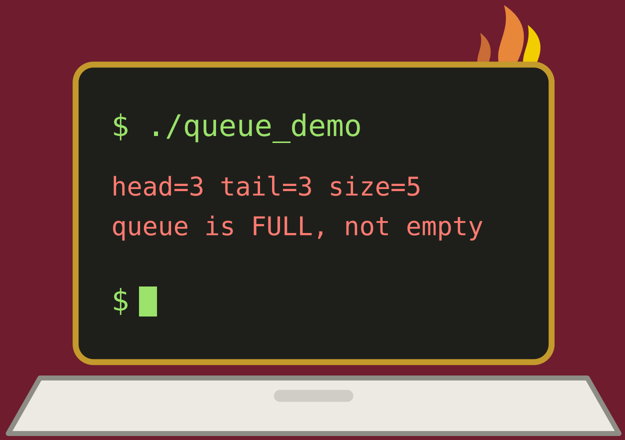
- Every operation O(1)
- No shifting, ever
- Almost no memory overhead

Costs

- Fiddly index arithmetic
- Full vs. empty needs care
- Fixed capacity unless you grow it

> A linked-list queue avoids the fiddliness, `add` at the tail, `remove` at the head, both O(1): at the cost of a pointer per item and a `new` on every insert.

Demo



Watching head and tail chase each other around.

(full and empty look identical. that is the whole bug.)

Deques

Pronounced "deck", a **d**ouble-**e**nded **que**ue.

```
#include <deque>

deque<int> d;
d.push_front(1);   d.push_back(2);
d.pop_front();     d.pop_back();
```

Push and pop at either end, all O(1). A deque can act as a stack *or* a queue, which is why the STL's `stack` and `queue` are built on one by default.

What If We Just Computed the Index?

A tree finds an item in O(lg n) by comparing. An array finds index `i` in O(1) by arithmetic.

> So: turn the *key itself* into an index. No comparisons, no walking, one calculation and you're there. That's a **hash table**.

A Hash Function

```
unsigned hash(const string &key, unsigned capacity) {
    unsigned sum = 0;
    for (char c : key) {
        sum = sum * 31 + c;          // mix the characters
    }
    return sum % capacity;           // fold into the table
}
```

- **Deterministic**: same key, same slot, every time
- **Fast**: otherwise you've lost the advantage
- **Spreads out**: keys should scatter across the table

> The `% capacity` is what forces an unbounded key space into a fixed number of slots, and that is exactly why collisions are unavoidable.

Collisions

`slot 0`		**(empty)**
`slot 1`	**"ada" → "bob"**	**both hashed to 1**
`slot 2`		**"grace"**

Two keys, one slot. With 4 billion possible strings and 100 slots, this is not bad luck; it's arithmetic.

Chaining: each slot holds a linked list of everything that landed there

Open addressing: on a clash, probe forward for the next free slot

TALK TO YOUR NEIGHBOR · TTYN

With chaining, what is the worst case for `find`?

A. O(1): hashing is always constant

B. O(lg n), the chain is a tree

C. O(n), every key could hash to the same slot

D. $O(n^2)$

Load Factor

```
load factor = items / slots
```

- Low (say 0.5): short chains, fast lookups, memory sitting empty
- High (say 5.0): memory well used, chains long, lookups slower
- Most implementations **rehash**: double the table and redistribute: around 0.75

> This is the **time–space trade-off** in its purest form: buy speed with empty slots. Rehashing is O(n), amortised across the inserts, exactly the doubling argument from week 5.

Where Hash Tables Sit

Operation	**Hash table**	**Balanced tree**	**Sorted array**
find	**O(1)** avg	O(lg n)	O(lg n)
insert	**O(1)** avg	O(lg n)	O(n)
remove	**O(1)** avg	O(lg n)	O(n)
sorted listing	O(n lg n)	**O(n)**	**O(n)**
worst case	O(n)	O(lg n)	O(lg n)

Fastest on average, and the only one with no sense of order. Every row of this table is a design decision you can now make on purpose.

Cost Summary

Structure	Add	Remove	Peek
Stack (array, top at end)	O(1)	O(1)	O(1)
Stack (list, top at front)	O(1)	O(1)	O(1)
Queue (naive array)	O(1)	**O(n)**	O(1)
Queue (circular array)	O(1)	O(1)	O(1)
Queue (linked list)	O(1)	O(1)	O(1)

> One row is worse than the others, and it's the one you'd write first without thinking. That's the lesson.

Week 09 Recap

- An **ADT** fixes the behaviour and leaves the storage open
- Stack = LIFO; Queue = FIFO
- Put the action where the structure is fast: end of an array, front of a list
- A naive array queue makes `remove()` O(n); a **circular array** fixes it
- `(i + 1) % capacity` is the whole trick
- `head == tail` is ambiguous, keep a size, or leave a gap

Next: a10 · a11 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**